



A Structured Approach to Developing Real-Time, Distributed Network Applications for ITS Deployment *

D.J. Dailey [†]; M.P. Haselkorn [‡]; D. Meyers [§]

INTRODUCTION

The management of large, complex systems, has been greatly aided by the use of increasingly sophisticated sensors and communication technologies used to obtain and process real-time data on system activity. In the realm of transportation, this type of system falls under the scope of Intelligent Transportation Systems (ITS). Typically, the data for ITS, management and information systems is generated by multiple types of sensors geographically distributed throughout the service area. These multiple types of sensors make measurements of various system parameters, each of which provides a partial picture of the current situation. In these types of applications, data typically moves from the sources, through some applications, to the ultimate end user. This type of data is used for multiple purposes, including: management and control, billing, end user information, planning, and emergency response.

As ITS sensors and communication infrastructure continue to advance, ITS software developers will increasingly find themselves working in distributed computing environments where there is the potential to access and combine multiple types of real-time data in order to deliver various services to distributed clients. These developers need to address a range of issues, including: (1) Sensors vary greatly in technology, complexity, and location; this implies a uniform method is needed to make data available on the network in a consistently interpretable form, and this

*This work was supported in part by a series of contracts with the Federal Highway Administration and the Washington State Department of Transportation

[†]Department of Electrical Engineering, University of Washington, Seattle WA 98195

[‡]Department of Technical Communication, University of Washington, Seattle WA 98195

[§]Department of Electrical Engineering, University of Washington, Seattle WA 98195

method must accommodate multiple technologies. (2) Access to data is necessary for numerous, geographically distributed users for multiple purposes; a single data source must be simultaneously accessible to various types of distributed clients. (3) The number and type of sensors changes over time, and new sensors will be added; data structure definitions change with sensor type, and the data structures used to interpret the data stream must track these changes. (4) A single type of data provides only a partial picture of the current state of a large system; multiple data sources are used with data fusion algorithms to estimate a system state. (5) Clients require access to data at geographically distributed sites. (6) Geographically distributed sensors are likely to be operated by, and located in, politically distinct jurisdictions. Further, data clients are likely to come from politically distinct organizations, necessitating reliable and flexible mechanisms for data security. (7) Data must be available in real time. (8) Software supporting these types of distributed systems must be reusable to leverage previous development efforts.

A distributed application can be defined as “An application program consisting of several cooperating components running on different physical nodes”[1]. In this paper, we describe a structured approach to developing ITS applications which are instantiated as real-time, distributed computing applications that are suitable for distributing dynamic data in real time to a large but authorized group of users. This approach effectively and economically addresses all the above issues. The approach is based on an asynchronous, distributed, client/server architecture and relies on the creation of autonomous, reusable pieces of software. Within this approach, applications are seen not as clients, but rather as configurations of the various structured software components.

Our structured approach is suitable for creating distributed applications that address a specific class of problem. The types of distributed applications for which our development environment is most appropriate are applications: (1) requiring multiple real-time data sources, (2) having unidirectional data flow, (3) having data whose content and structure may vary slowly (on the order of minutes) with time, (4) required to scale to support thousands of clients, (5) requiring each client to be identified as an authorized consumer of the data, and (6) requiring real-time data on a time scale of the order of tenth’s of seconds. These features are typically present in ITS applications where data from a variety of remote sensors are combined to provide surveillance or control functions. However, this general class of problems is becoming more prevalent with the widespread access to the Internet and the development of new sensor technologies.

It is noteworthy that the class of problems addressed by this approach differ substantially from the typical distributed database problems. The table security, audit, serialization and concurrency problems often addressed in the distributed database problem are absent, and the system reliability and fault tolerance are addressed using the monitoring tools described later in the paper.

In this paper we first present our general development framework and then demon-

strate our approach by describing several applications we have created and deployed in the domain of Intelligent Transportation System (ITS) (It is noteworthy that our paradigms fit within the context of the FHWA ITS Architecture [2][3]).

THE DEVELOPMENT ENVIRONMENT

In this section we describe a development environment designed to address the issues faced by ITS developers. In this environment: (1) real-time data is asynchronously available from various geographically distributed and technologically independent sources, (2) real-time data is provided to various geographically separated clients on an as-needed basis, (3) real-time data fusion is used to produce additional “value-added” data streams, (4) clients are added through an efficient scaling mechanism that maintains desired security, and (5) the data dictionary of the real-time data changes slowly over time, and these changes are handled in a principled manner.

Our development environment consists of four components, and we define an application as an appropriate configuration of the components. The components constitute a toolkit that implicitly handles messages and communication, freeing developers to deal with the flow and interpretation of data streams.

Components

In this section we provide an overview of the functions of each component type. The “Applications from Components” section presented later describes how components are linked to form applications.

Our development environment is composed of software components that can be configured and linked to construct applications. Each component is: (1) autonomous, i.e. components are distributed across computing platforms; (2) independent, i.e. components can operate in parallel; (3) a peer of any other component, i.e. components can be placed into an evolving hierarchical matrix with great flexibility; and (4) reusable, i.e. coding is efficient and much of application design is reduced to configuring available components.

There are four types of components, each of which affects the flow of data (See Figure 1). The component types are: (1) a Source Component which makes a data stream available, (2) a Redistributor Component which obtains a data stream from one component and redistributes it to one or more other components, (3) an Operator Component which obtains data streams from one or more components and creates a new data stream for distribution to one or more components, and (4) a Sink Component which obtains data streams from one or more components. Redistributors and Operators function both as client and server, while Sources are servers and sinks are clients.

We use the existing network inter-process communication (IPC) facilities as a base for implementing the components that make up the building blocks of our application

development environment. Unlike previous work that required extensive software for each participating operating environment, we require only a socket interface to an IPC library to implement applications on a wide variety of operating systems at low cost. Our approach uses three IPC mechanisms: (1) the ability to connect to a specific service at a specific location on the network, (2) the ability to answer requests for service from remote clients, and (3) the ability to maintain connections between the components of our applications for the purposes of data transfer. Implicit in the use of the IPC services is the expectation that the appropriate guarantee of delivery and reassembly are met. In the examples presented later in the paper the transmission control protocol/internet protocol (tcp/ip) are used to provide the transport layer services implicit in the use of ICP services. The use of a network wide transport protocol implies that the application can be widely distributed geographically while guaranteeing connectivity. Problems associated with interoperability, compatibility, and addressing are invisible to the applications discussed here because the transport mechanisms supporting the IPC paradigm addresses these issues directly. Access security and connectivity problems are addressed in the following section.

Elements of a Component Before describing these four components in greater detail, we first describe three elements which are key, reusable pieces of these components. Three of our four components (Source, Redistributor, and Operator) are built from three specific reusable elements. The elements and a state diagram for each element are shown in Figure 2.

The “Connection Daemon,” element waits for and accepts asynchronous IPC requests across a network. These requests are for either data or administrative functions. It services these requests by first establishing the authorization of the requester and then either accepting a connection from a client requesting data or executing an administrative function such as restart. Once network connectivity and authentication are established the information about the IPC connection to the client is passed to the “Output Multiplexer.”

The “Output Multiplexer” element maintains a list of connected clients and multiplexes the data stream to all active clients. It also maintains state information in the form of a data dictionary structure which is the first information passed to a newly connected client. This data dictionary definition allows the client to interpret the data stream. The data stream to be multiplexed is received asynchronously from the element labeled “Input Buffer.”

The “Input Buffer” initiates and maintains connectivity with a data source as well as receiving available data from a data source. This element provides a data buffer for the data from a source, and forwards the data to the “Output Multiplexer.” This process guarantees that the component remains synchronized with the data stream from a data source. This synchronization is necessary because the data stream from sensors is sent in a frame and each frame has a fixed length (defined in the beginning of the frame). If synchronization is lost, this element terminates the connection to

the data sources and reestablishes the connection to re-synchronize the data stream. Multiple input buffers may exist in a single component to receive data from several sources.

The Connection Daemon, Input Buffer, and Output Multiplexer all exist on one computer, communicate with each other asynchronously, and operate in parallel. This parallelism is obtained using operating system support for multiple processes which may time slice one CPU or execute each process on parallel CPU's. The parallel operation of these elements is necessary due to the asynchronous nature of the tasks performed by each element. The elements: (1) Connection Daemon, (2) Input Buffer, and (3) Output Multiplexer, make up the heart of the reusable code in our approach, and three of the four components, the Operator, Redistributor, and Source, use these three elements as the building blocks. Following is a more detailed description of each component type and how they are constructed.

Source A Source component exists for each sensor technology and makes the data stream from that technology available on the network. It does this by: (1) interfacing with the sensor technology, (2) using knowledge of the content of the data stream to build a self-describing data dictionary into that data stream, and (3) making the data stream available to other components on the network. Note that the Source component does not include the technology which generates the data stream, but rather is a tool that allows developers to ignore the nature of that technology. Sensor technologies range from a single sensor with a single, simple data type to computer systems providing rich streams of complex data. Whatever the nature and complexity of the data generating technology, the related Source component packages its stream of data to give it a peer status with other data streams on the network.

The Source component type adds one element to the three basic elements just described, namely a process specific to the sensor/system that is the originator of the data stream (See Figure 3). This additional process communicates directly with the sensor/system to obtain data and information about the format, contents and meaning of the data stream. It builds data structures—to define format, contents and meaning—that are provided to clients as an initial data dictionary before transmitting the sensor data. These constructs produce a self-defining data stream with an initial dictionary and following data. As the content of the information coming from the sensor changes in format, content and structure, this component builds a new data dictionary and passes it on to “downstream” components.

Redistributor A Redistributor takes data from one component and multiplexes it to other components. Each Redistributor is client to a particular data-providing component and allows the data stream from that component to “fan out” to each of its own client components. The Redistributor component has two basic functions: (1) scaling and (2) authorization.

In our structured development environment, scaling is achieved by linking Redistributors in a hierarchical fashion. Since each Redistributor services N clients, M layers of redistributors service $N \times M$ clients. Because components are autonomous, redistributors can reside on any computer and operate in parallel. In this way, scaling can be economically achieved in fixed-size blocks through the addition of redistributors operating on parallel computing platforms (See Figure 4).

The redistributor also performs authorization for data access. The redistributor maintains a dynamic client list, and whenever new data is available, passes it on to clients on that list. When a request for data comes from a component not currently on the client list, a check is made against an authorization list. This authorization list is part of the initialization information for each redistributor and may remain as simple as a static list or may include dynamic queries to a remote database. The dynamic or static authorization list and authorization mechanism are part of the configuration information. Similar authorization is performed by Operator and Source components as well.

The Redistributor component type is constructed from the three elements described above as well as configuration information that includes the data source network location and authorization information listing allowable clients. As such, Figure 5 and a table of configuration information represent the Redistributor component type.

Operator The Operator component type is comprised of the three basic elements plus a data fusion activity that operates on the data content as it is passed from the input buffer to the output multiplexor (See Figure 6). An Operator gets data streams from one or more components, performs functions on those data streams, and makes available a single newly-created data stream to one or more components. Its general purpose is “data fusion,” converting data from the form provided by the serving components to a form desired by the eventual clients.

Each Operator component is associated with a specific data fusion activity. These activities can range from the combining of like types of data to the creation of new information from a variety of data sources. Like the Redistributor, the Operator also performs authorization for data access.

Sink Our Sink component is the portion of our approach that is commonly viewed as an application by a user. It is the component that typically either has a user interface, or performs an activity that effects the environment. It typically obtains data from one or more components (usually an Operator or a Redistributor which is downstream of an Operator). Examples of several types of Sinks are provided in the “Example ITS Applications” section.

These four component types: (1) Source, (2) Sink, (3) Redistributor, and (4) Operator—are the basis for creating our distributed applications as described in the next section.

Applications from Components

In our approach, applications are constructed from components by connecting Sources, Redistributors, Operators, and Sinks in a hierarchical structure. We define an application to be composed of all of the structures necessary to take the original sources of data, perform data fusion, and distribute the resulting information to the data sinks. These structures include the individual component types, structural information about the path for data flow through the application, and information necessary to allow the Operators and the sinks to use the data.

For example, the simplest application is a single client single server model in which a data sink operates on the original source data (e.g. the data sink displays the sensor status or output). This application is composed of two components, a Source and a Sink. The elements of these two components use: (1) connection information allowing the Sink to connect to the Source, (2) authorization information allowing the Source to send the data to the Sink, and (3) a shared definition of the interpretation of the data stream. Such an application is shown in the shaded area labeled (1) in Figure 7 and is similar to the single client, single server notion of distributed applications. However, in our approach, most applications are composed of numerous components.

A common application scenario would include: (1) multiple data specific Sources that are geographically and technically disparate, (2) a Redistributor for each Source, (3) an Operator for each type of data, (4) an Operator for each type of information that is to be created from available data products, (5) a Redistributor for each block of Sinks, and (6) multiple special purpose Sinks that require access to data streams to achieve their purpose. Applications are composed of several of our base components and are initiated using remote procedure calls that start individual components on local or remote computers and initialize these components with their data sources.

Implicit in any application design that requires the data to flow through several components serially in the single point of failure problem (e.g. all the components of a particular application must be functioning for the application to function). However, the individual components functionality is monitored by the external tools described in the next section, and this monitoring scheme in conjunction with the user interface tool described in the next section provide the basis to ensure reliable operation of applications built using the paradigm presented here.

The “Example ITS Applications” section of this paper presents some illustrative examples of applications we have built using the paradigm presented here. We discuss tools for application creation and management in the next section.

Tools for Application Creation and Monitoring

We have created a set of tools to be used within our framework to create and manage applications. The applications are created and managed using a set of graphical tools to assemble the hierarchy of components that make up an application, and the same

type of graphical tool is used to identify the status of elements within operating components.

To support the implementation of management tools, each of the elements within a component has monitoring facilities that report status to a “simple control and monitoring daemon” (SCMPD) over a connectionless channel. This status consists of both static data such as: (1) the Source and Sink ports and (2) the address of the server from which it receives data, as well as dynamic data such as: (1) the time of the last message processed by the component, (2) status of the component (such as a heartbeat indicator), (3) the present state of the component (initializing, running, shutting down), and (4) the time of the most recent change of state. The SCMP daemon logs this information and also supports connections by a console application.

The console application creates a connection to a SCMP daemon and produces two windows; the first is an acyclic planar graph representing the connections between the components making up the applications that are reporting to this SCMP daemon, and the second window provides a detailed accounting of the status of the elements within the components reporting to that SCMP daemon. Examples of these windows are shown in Figure ???. The console application also permits either manual or automated responses to the components in the case of error conditions being detected. In most cases the report of a serious error or the loss of heartbeat from an operating component results in that instantiation of the component being destroyed and a new instantiation created to restore the end to end service necessary for the operation of applications that use the component construction paradigm.

An application similar to the console is used to create the control tables that instantiate each of the components and create the connections between the components. An application is created by starting a set of components that have the inter-component communication defined in terms of the Source and Sink address and port numbers. Examples of ITS applications created using this approach are presented in the next section.

Example ITS Applications

To illustrate an implementation of our approach, we describe several Intelligent Transportation System applications we have constructed using our paradigm. ITS applications are good examples of the type of remote sensor problems for which our approach is appropriate. In ITS applications there are typically large numbers of sensors that are geographically diverse and controlled by autonomous political and fiscal entities. However, the applications need access to data from a variety of sensors widely deployed and owned by other entities. These sensors typically have update rates on the order of milliseconds to seconds. Two examples of ITS data sources are congestion monitoring systems and probe vehicles.

Congestion monitoring is implemented using a variety of sensor types. In our demonstration applications we use two types of sensors, inductance loop sensors and

CCTV cameras. The inductance loop sensors are hundreds of wire loops buried in freeways and arterials that have associated electronics that can identify properties of vehicles passing over the loops. These properties include presence of a vehicle, duration a vehicle is present, count-per-unit time, average speed of vehicle, length of vehicle, and type of vehicle. The properties measured vary based on the available electronics. The CCTV cameras provide a video image of the roadways at a variety of locations throughout a large geographic area.

In our demonstration applications we also use probe vehicles. Probe vehicles are located in real time to provide a second by second tracking of the vehicle fleet. This type of fleet management is done by both the public and private sector (e.g. transit management or delivery fleet management).

In the next two sections we use two example ITS applications to describe the data flow and parallelism of operation of the servers. The two applications are: (1) an Advanced Traveler Information System (ATIS), named *TrafNet*, that provides real time congestion, speed, and travel time information about the regional freeway system, and (2) a real time Advanced Public Transportation System (APTS) application, named *BusView*, that provides real time transit vehicle locations as well as schedule information. We describe our instantiation of servers from top to bottom, which is the direction of data flow in Figure 8. In this Figure the parallelism between the server structures to support the two applications to be described is clearly visible in Figure 8, with *TrafNet* on the left and *BusView* on the right.

TrafNet We have used two types of data sources (probe vehicles and inductance loop data) to build a variety of applications using our development environment. The first example is an application we call *TrafNet*, which provides real-time updates of the traffic conditions to Internet users.¹ By way of example we describe the data flow and architecture of *TrafNet*. In Figure 8 the left side of the figure represents the architecture of *TrafNet*, and the data source is a *Traffic Management System* (TMS) operated by the Washington State Department of Transportation (WSDOT) which polls several hundred Inductance Loops every 20 seconds. WSDOT, who funds the collection of this data, uses it for traffic management; however, this type of information has potential value to travelers and shippers as well. We make this data, and products derived from this data, available on the Internet using the paradigms proposed here.

The Source component in this application eavesdrops on the loop data while the loops are being polled and maintains a data dictionary that describes the structure of the loop data. This structure includes information about loop locations and loop type (e.g. the type of measurement and units of the measurement). The Source is connected to a network and responds to requests for loop data by first sending the data dictionary and then following that with the updated set of sensor values as soon

¹The executable for *TrafNet* can be obtained from <http://www.its.washington.edu/trafnet>.

as they become available. This process is actually implemented on the computer which runs the WSDOT TMS system and is named *TMSUW*.

The next component, labeled *Loops_forward*, is an Operator which receives the data dictionary and data from *TMSUW*. To this data it adds information about the route and milepost marker location for each of the items found in the data dictionary, and as such this Operator component is engaged in “data fusion.” This component operates on a computer that is physically located at WSDOT’s Traffic Systems Management Center (TSMC), and the significance of the physical location is that this data fusion server can control what data is allowed (by agreement with WSDOT staff) to leave WSDOT’s premises. This control over access to the data provides agency autonomy and network security. There are two firewalls between the TSMC subnet and the Internet that protect WSDOT internal computers from the Internet community. The firewalls allow only specific computers at specific TCP ports to connect to this computer.

The component labeled *Loop_Fanout* in Figure 8 is a Redistributor component that makes the loop data and data dictionary available to a larger number of clients. It maintains connectivity with *Loops_forward* while accepting and authenticating requests for connections for data. This is the last component in the hierarchy where the original data as recorded by the sensors is available; downstream servers modify the original data stream.

The component labeled *Loop_Server* is an Operator that performs data fusion by adding position in latitude and longitude to the data dictionary based on the loop identifier and the mile post information. It further scales the data to known units (e.g. vehicle counts converted to vehicles per hour).

The next server component labeled *TRServer* is another Operator component that combines data from individual loops to make congestion estimates and packages the resulting information in a specific format and order to be used by the presentation Sink component labeled *TrafNet*. The connection between this server and the *TrafNet* sink can potentially be over slower media so this component minimizes the amount of data that need be sent per update of the *TrafNet* screen.

To make the *TrafNet* data stream available to large numbers of users, we can add redistributors downstream of this component; however, to date, that has not been necessary.

The *TrafNet* program (the sink at the bottom) is a MicroSoft Windows 3.1 application that uses the Winsock network interface. The end user looking at the *TrafNet* screen sees only the graphic representation of the freeway system with congestion indications updated every 20 seconds. The user can select entrance and exit ramps to request information about specific trips. This is a first example of how our approach facilitates creating value-added applications as well as making intermediate forms of the data available for other uses.

BusView A second example of a distributed application created within our framework is BusView, which uses probe vehicles in the form of transit coaches to create an Advanced Traveler Information System (ATIS) application that displays the location of the transit coaches on a digital map in real time.

BusView, shown in the right-hand column in Figure 8, uses a transit carrier's automatic vehicle location system as the sensor for the Source component. In this case the Source component (labeled *AVLUW*) eavesdrops on communication between components of a proprietary AVL system used by the transit carrier's operations staff to measure schedule adherence and provide the traffic managers a location in the event of an incident. The data in this case is distance along a planned path, which, with knowledge of the routes and several coordinate transformations, can be used to estimate location.

Like the component *Loops_forward* in the previous example, *AVL_AVLUW* is an operator located on a computer in the AVL operations center of the transit carrier. And like the previous example, it limits the information that can be sent out of the agency. In this case the numerical identification of the driver is in the data received by *AVLUW* but is removed for privacy reasons before the data is allowed out of the physical control of the transit agency. This component is also behind the firewalls protecting the transit AVL system from Internet users. Once again this Operator component provides for network security and agency autonomy, as well as removing information from the data stream.

The raw data, a distance along a known path, is multiplexed by a Redistributor (labeled *AVL_Rebroadcast*). Like the component *Loops_fanout* in the previous example, *AVL_Rebroadcast* multiplexes the original data to make this raw data widely available.

The data from *AVL_Rebroadcast* is passed to an Operator (labeled *AVL_Trip*) which uses information in the AVL data packet to associate a specific vehicle with a specific trip in the transit schedule. The data from the AVL system arrives at an average rate of 11 coaches per second, and a 20,000 record database of scheduled trips is searched for each coach to obtain this schedule correspondence. This trip information is added to the record for each coach and made available to the downstream clients.

The data from *AVL_Trip* is passed to an Operator (labeled *AVL_Position*) which can use the data, information about planned routes, digital maps, and coordinate transformations to calculate a latitude and longitude value for the transit (probe) vehicles. The calculation of position requires: (1) identifying the particular coordinate lists in one of 2500+ files that represent every possible vehicle route, (2) selecting the segment within the list on which the vehicle is estimated to exist, and (3) using a Newton's method to perform an inverse Lambert projection from the proprietary coordinate system of the AVL system, at a rate of 11 vehicles per second. The ability for a component to operate on independent CPU's designed into our approach means that we can select a CPU of appropriate power for each of the individual tasks. This Operator, for example, takes the entire resources of one computer to keep up with

the data flow, and as such does not coexist on a CPU with the other Operators in these examples.

Downstream of the *AVL_Position* server is a Redistributor labeled *AVL_Fanout* which multiplexes the data stream containing the vehicle positions to a variety of users. This Redistributor frees the upstream process from multiplexing in addition to performing the positioning solution.

The server component labeled *AVL_Ivhs* creates a very customized data stream for the *BusView* presentation. Like *TRServer* in the previous example, the connection between this server and the *BusView* sink can potentially be over slower media. This component minimizes the amount of data that need be sent per update of each transit vehicle on the *BusView* screen.²

A digital map and the vehicle location information, along with schedule information are displayed graphically on the *BusView* screen creating an information system for transit riders. *BusView* is a second example of using our approach to create distributed applications that use real-time data from a single source. In the next section we describe a hybrid system that takes data from both of the server stacks just described.

Hybrid Example: SWIFT A third example instantiation of our paradigms is a hybrid of the previous two applications. This hybrid application is Seattle Wide-area Information for Travelers (SWIFT), a Federal Highway Administration (FHWA) sponsored operational test. This project combines data from the sources used in both Trafnet and BusView to create new information. Traffic congestion and probe vehicle information described above are used to estimate speed (and travel times) on freeway links, and that information is provided in turn to a commercial wireless carrier who broadcasts it to subscribers. These subscribers receive the information on three principle devices: (1) a Seiko ReceptorTM, (2) a Delco car radio, and (3) an IBM portable computer. Each of the receiving platforms represents a potential value-added resale of the information. Figure 9 shows the collection of components that create the information source for the SWIFT application. Note that many of the components that were part of the previous two applications are also used in the SWIFT application, and this simultaneous usage does not effect the operation of the previous two applications.

Conclusion

The approach to ITS development described in this paper embodies several specific features. These include: (1) an architecture that delivers real-time data to a large number of consumers while maintaining a mechanism for authorization, (2) a definition of distributed application that includes an entire set of components arranged in

²Request for access to the BusView URL can be sent to busview@its.washington.edu.

a hierarchical structure, and (3) a clear mechanism for building “value-added” applications that use a generally available data stream. In following this approach ITS developers are not only addressing the numerous issues facing them today, but also laying the foundation for intelligent transportation systems of the future.

References

- [1] A. Schill. Distributed application support: Survey and synthesis of existing approaches. *Information and Software Technology*, 32(8):545–558, 1990.
- [2] Rockwell International Joint Architecture Team, Loral Federal Syatems. Theory of operations. *ITS Architecture*, October 1995.
- [3] Rockwell International Joint Architecture Team, Loral Federal Syatems. Physical architecture. *ITS Architecture*, October 1995.

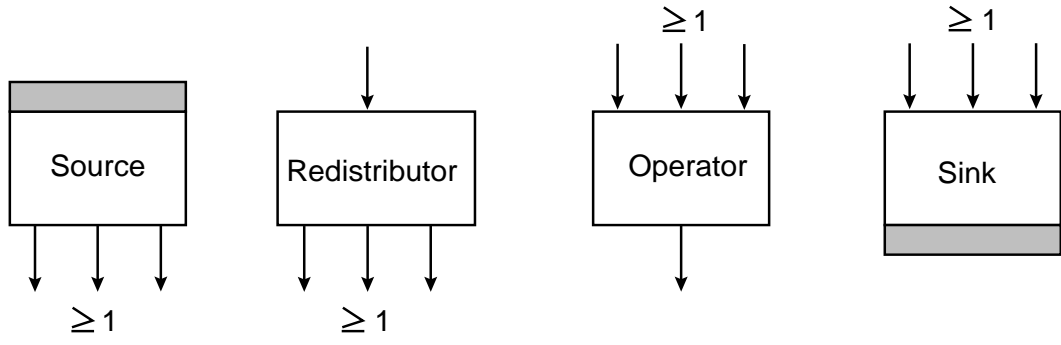


Figure 1: Components

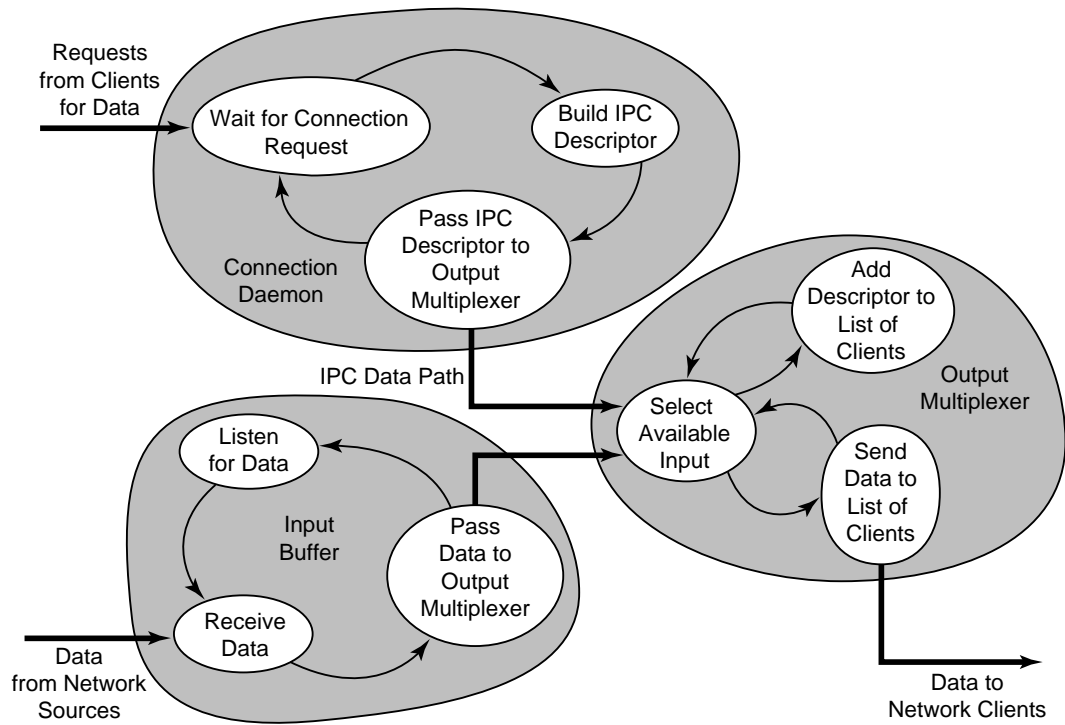


Figure 2: State Diagrams for the Elements in a Redistributor

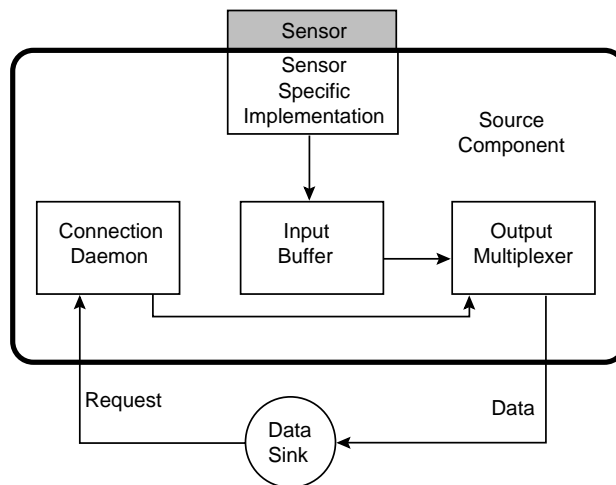


Figure 3: Elements of Source Component

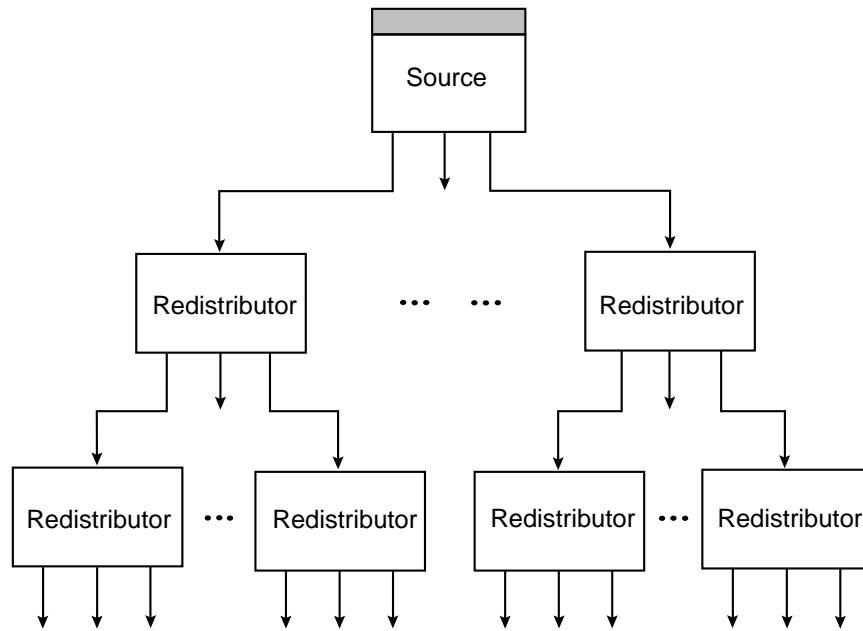


Figure 4: Hierarchical Scaling Mechanism

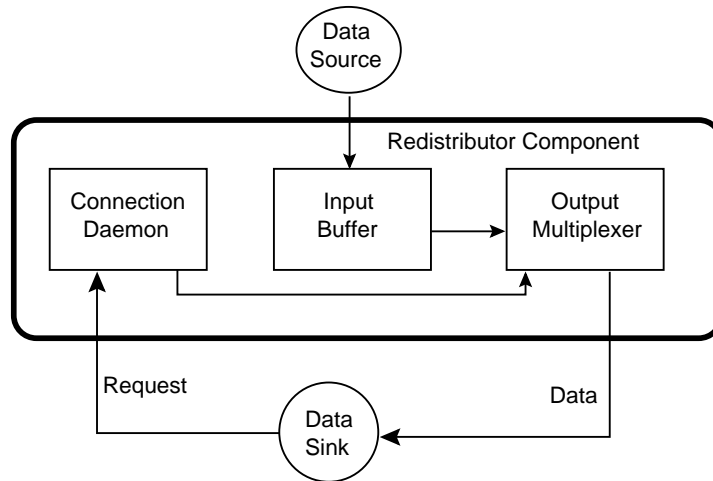


Figure 5: Redistributor Component Elements

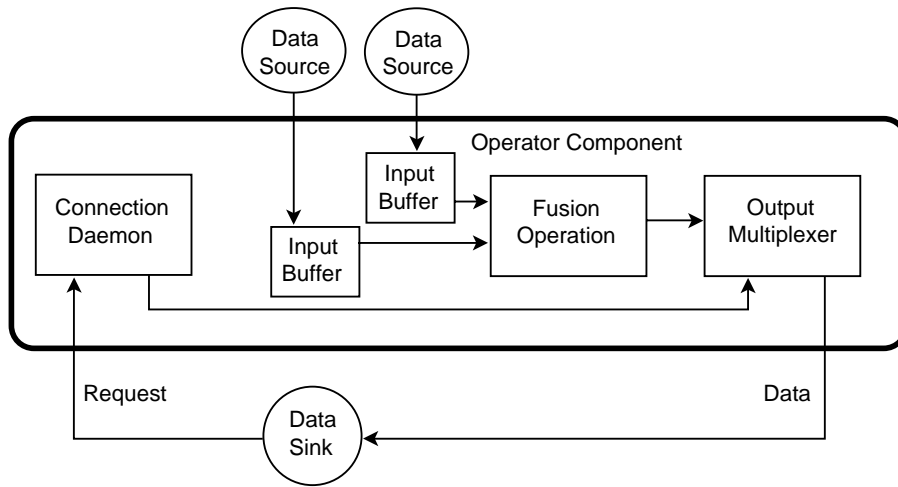


Figure 6: Elements of the Operator Component

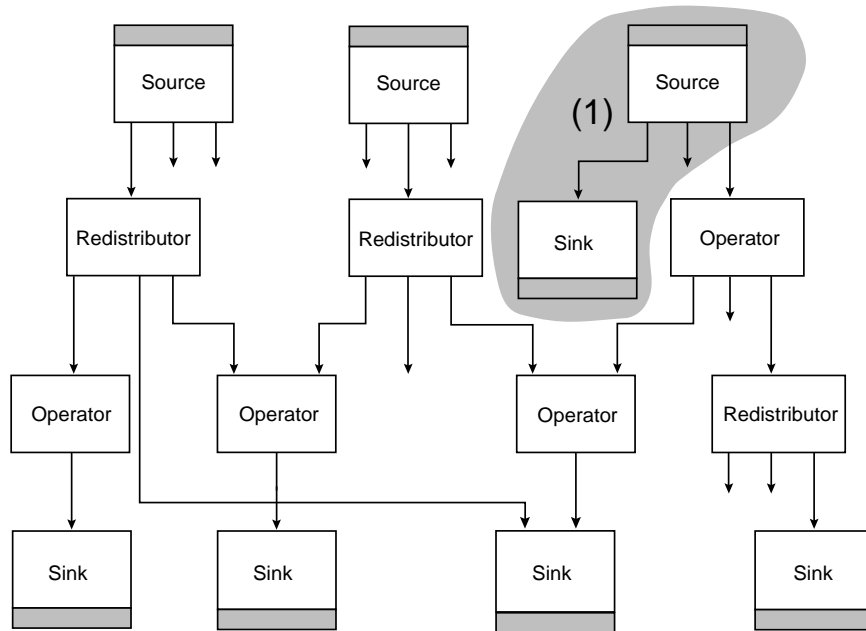


Figure 7: Construction of Applications from Components

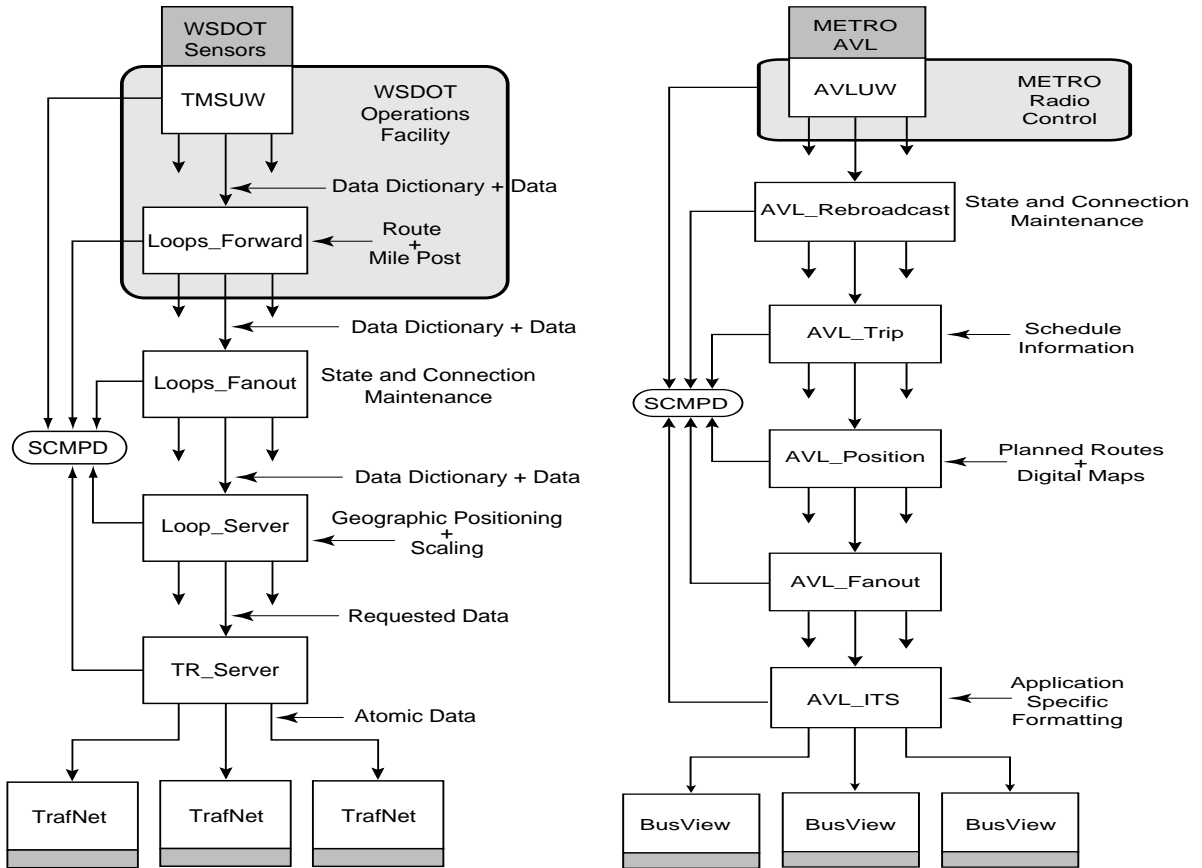


Figure 8: TrafNet and BusView Applications

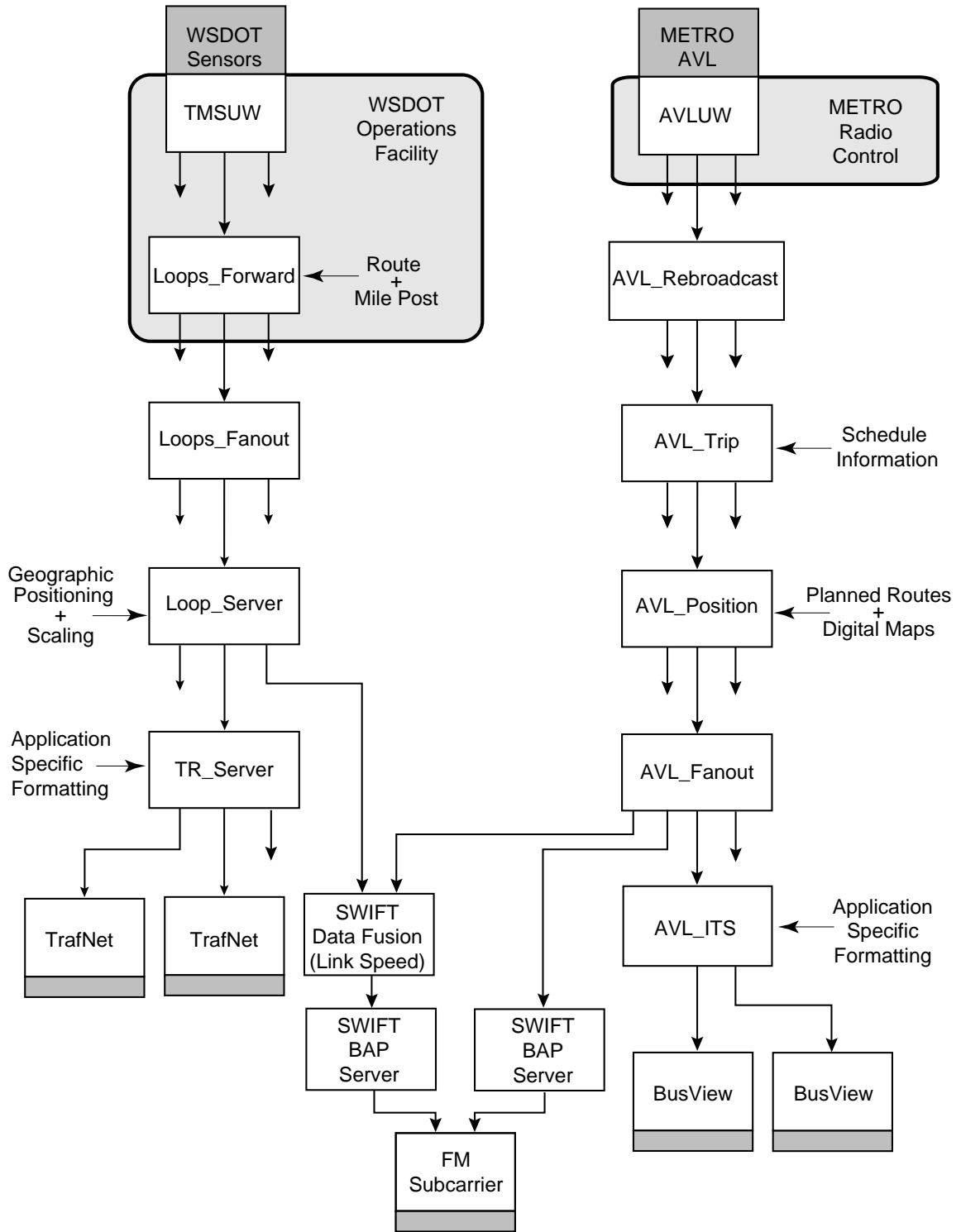


Figure 9: SWIFT Application in the Context of TrafNet and BusView.